Optimisation of Neural Network Architecture

Henry Twist

April 5, 2019

Abstract

Neural networks have become a useful tool in a great deal of modern day software and their utility is ever growing. Constructing neural networks requires various design choices such as their architecture and some parameters for the training process. Optimising these hyperparameters is a complex problem due to the strange domain that they lie in and the absence of feasible derivatives, which causes traditional optimisation methods to fall short. This then leads to ad hoc experimentation being used which is clearly not an ideal solution. In this paper we will explore the theory behind the concept of neural networks, how to train them and then approach optimising their architecture. Using Bayesian optimisation, posterior distributions can be formulated to make predictions about network performance with different architectures which guides the optimisation process.

Contents

1 Introduction

Artificial intelligence (AI) is not a new discovery and started as far back as the time of Alan Turing. However, the term was not coined until 1956 in a conference set up by computer scientist John McCarthy, who is well known for his research in this area. Recently it seems that AI has become increasingly popular, particularly in the public eye, and has become an interest point for giant conglomerates. This has been exacerbated by the uses outside of the scientific communities which have only just come to fruition: security, the IoT (internet of things) and medicine being the obvious examples. Furthermore, as AI is lending itself to more commercial utility, technology is being developed specifically to cater for running these 'intelligent' algorithms. For example, Tesla are starting to use proprietary GPUs for use in their newer cars and even Apple have started incorporating a dedicated chip to optimise AI performance in their smartphones.

1.1 Machine Learning and Neural Networks

First it is probably necessary to introduce *machine learning*, a specific application of AI that involves machines being given data with an intent to somewhat learn from it. What learning entails is contextual and is often split into two classes: supervised and unsupervised learning, although there are many more distinctions. Here however we will restrict our analysis to *supervised learning* which involves learning some sort of mapping from input points to output points. These output points can be real, which is then known as regression; or categorical, which is then known as classification. We can weakly formalise the problem below.

Problem 1. *Given a training set of input/output pairs* $\{(\boldsymbol{x_i}, \boldsymbol{y_i})\}_{i=1}^{N}$*, approximate a function* $\hat{\boldsymbol{f}}$ *such that* $\boldsymbol{y_i} \approx \hat{\boldsymbol{f}}(\boldsymbol{x_i})$ *for* $i \in \{1, \dots, N\}$*.*

In this context however, a function approximation is not enough. The goal is to actually predict unseen data points and fitting a function to the known data would not necessarily be the best solution. It is very easy with a problem like this to *overfit* the data, which is essentially learning the data and not the patterns surrounding it. Avoiding this is a fine art, but can be automated to some extent which is what we will be discussing here.

Now we can move to talk about neural networks. Actually a *neural network* broadly refers to any circuit of neurons, but here we will only be referring to artificial neural networks and will drop the "artificial" prefix. The term is loosely inspired by neurons in an organic brain and the way that synapses transmit a signal from one neuron to another for processing. In the same way organic neural networks do, artificial ones have some sort of architecture or structure of these neurons, where each one takes input from all the previous neurons and acts as a function on these inputs. The main idea is then to optimise these 'functions' to fit the training data as outlined in Problem 1 and this is known as *training* the network.

A subject of contention is how to choose certain starting parameters of the neural network, more specifically parameters required before training. The combination of the architectural parameters and some more from the internal workings of the algorithm results in a very strange domain to optimise over with intractable derivatives. Not to mention that observations from a neural network are inundated with noise when trained, all of which makes optimisation with traditional methods difficult. Even if these issues were overcome, training a network with just one set of parameter combinations takes a massive amount of time which would make it impractical to repeat many times over. Frequently ad hoc experimentation is used to 'solve' this problem which apart from being inefficient and inaccurate,

can often lead to overfitting. Therefore we have a new problem outlined below.

Problem 2. *Given a neural network with undetermined starting parameters, efficiently optimise it to best solve a problem without overfitting the data.*

In this paper the goal is to approach neural networks from the beginning; go over the theory behind them; how to train them and then try to solve Problem 2 using Bayesian optimisation.

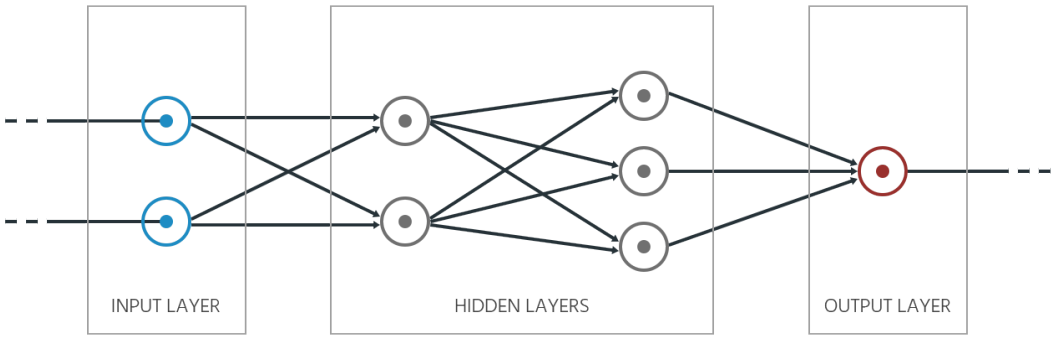


Figure 1: A simple neural network example.

2 Training Neural Networks

2.1 The Architecture

We can now start to introduce neural networks properly and formalise training them. The material in this section is based upon Higham and Higham [2] but expanded in scope and detailed more rigorously. Neural networks are made up of layers of neurons which we will call the *architecture*. To visualise the set-up of a typical network we can look at Figure 1 where we have 4 layers, with a neuron pattern of 2, 2, 3, 1. The first layer is called the *input layer*, where the number of neurons needs to be the same as the dimension of the input vector. The input layer does not actually perform any transformations on the data but acts more as a representation of feeding the input vector to the next layer of neurons. The layers in the middle are called the *hidden layers* and they accept input from the previous layer, apply a function and then pass the result to the next layer. Finally there is the last layer, the *output layer*, which needs to have identical dimension as the desired output. This layer transforms its input as in the hidden layers and then the output from this layer is the final output of the network. As seen in Figure 1 each neuron receives output from every neuron in the previous layer. To formalise this, we can introduce some notation for the problem:

- $\boldsymbol{x}$ - an input vector
- $\boldsymbol{y}(\boldsymbol{x})$ - the true output corresponding to $\boldsymbol{x}$
- L - the number of layers in the network
- n_l - the number of neurons at layer l
- $\boldsymbol{g}^{[l]}$ - the function representing the l^{th} layer
- $\boldsymbol{a}^{[l]}(\boldsymbol{x})$ - the output from layer l with initial input $\boldsymbol{x}$.

Using this notation and considering the properties of a neural network, for an input vector $\boldsymbol{x} \in \mathbb{R}^{n_1}$ we have that

$$\boldsymbol{a}^{[l]}(\boldsymbol{x}) = \begin{cases} \boldsymbol{x} & l = 1 \\ \boldsymbol{g}^{[l]}(\boldsymbol{a}^{[l-1]}(\boldsymbol{x})) & l \in \{2, \ldots, L\} \end{cases}. \tag{1}$$

It is worthwhile to note that due to the nature that the input is passed through the network, the functions $\boldsymbol{g}^{[l]}$ map from $\mathbb{R}^{n_{l-1}}$ to $\mathbb{R}^{n_l}$. Training the network is simple in concept and involves optimising these layer functions $\boldsymbol{g}^{[l]}$ so that the final outputs from the network best match the true output. So informally we are looking for each pair $(\boldsymbol{x}, \boldsymbol{a}^{[L]}(\boldsymbol{x}))$ to approximate $(\boldsymbol{x}, \boldsymbol{y}(\boldsymbol{x}))$.

2.2 The Layers

Now let's consider the actual functions that define the behaviour of the layers in the network. First define

$$\boldsymbol{z}^{[l]}(\boldsymbol{x}) = W^{[l]}\boldsymbol{a}^{[l-1]}(\boldsymbol{x}) + \boldsymbol{b}^{[l]} \quad \text{for } W^{[l]} \in \mathbb{R}^{n_l \times n_{l-1}},\ \boldsymbol{b}^{[l]} \in \mathbb{R}^{n_l},\ l \in \{2, \dots, L\}. \tag{2}$$

Then the functions at each layer take the form

$$\boldsymbol{g}^{[l]}(\boldsymbol{a}^{[l-1]}(\boldsymbol{x})) = \alpha(\boldsymbol{z}^{[l]}(\boldsymbol{x})) = \alpha(W^{[l]}\boldsymbol{a}^{[l-1]}(\boldsymbol{x}) + \boldsymbol{b}^{[l]}) \quad \text{for } l \in \{2, \dots, L\}. \tag{3}$$

Here $W^{[l]}, \boldsymbol{b}^{[l]}$ are unknown and are determined as a result of training the network, however we will omit the dependence on these parameters for brevity. The core part of this function should be quite familiar and looks like a form of multidimensional linear regression which is mostly all it entails. However, there is also a function α which is called the *activation function*. We will not be covering this in much detail here, but the choice of activation function has a big effect on training. For example, we could consider something called the ReLU (Rectified Linear Unit) activation function which is defined $R(z) = \max(0,\, z)$, which effectively restricts the calculations to a positive domain. Though in this analysis we will be considering the *sigmoid function* (a special case of the logistic function) which is defined

$$\sigma : \mathbb{R} \to \mathbb{R}, \qquad \sigma(z) = \frac{1}{1 + e^{-z}}.$$

This function has a very characteristic S-shaped curve as shown in Figure 2. There are a few properties which make this choice desirable. Its finite range makes training much more stable and it possesses an easy to work with derivative which will be useful in Section 2.4. Also its range makes it perfect for classification, as it provides an analogue version of a binary function which can represent whether something belongs to a particular class. For use later on, let us intuitively specify the vector analogue of the sigmoid function:

$$\boldsymbol{\sigma} : \mathbb{R}^d \to \mathbb{R}^d, \qquad \boldsymbol{\sigma}(\boldsymbol{x})_i = \sigma(x_i).$$

Now with this notation and a general understanding of what neural networks do at each layer, we can move on to training them.

2.3 The Cost Function

Training in this context is effectively solving Problem 1, and the function we are approximating can be viewed as

$$\hat{\boldsymbol{f}}(\boldsymbol{x}) = (\boldsymbol{g}^{[L]} \circ \cdots \circ \boldsymbol{g}^{[2]})(\boldsymbol{x}).$$

Therefore to solve Problem 1 we are required to choose $W^{[i]}, \boldsymbol{b}^{[i]}$ such that $\hat{\boldsymbol{f}}(\boldsymbol{x})$ above best fits the data. To do this we must quantify this fit, hence we will define a cost function or error function to minimise. Formalising some notation, we will say that there are N training

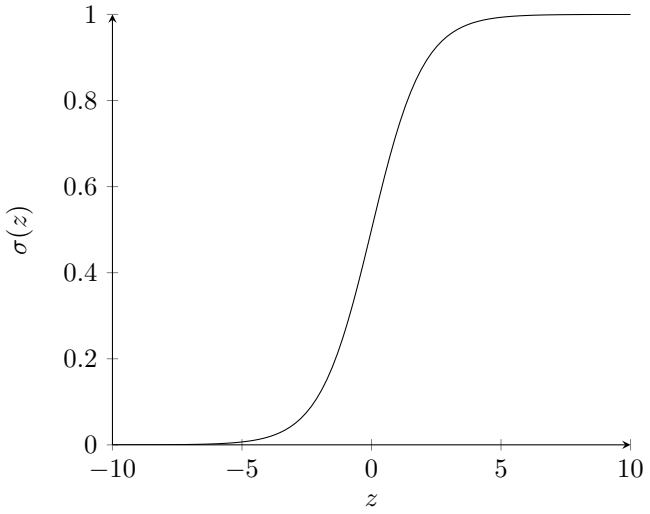


Figure 2: The sigmoid function.

points $(\boldsymbol{x_i})_{i=1}^N$. In fact generally the data you have available is split into training data and test data, so there is something to test the network on after training as detailed in Section 2.5. Ignoring the test data for now, as it does not affect training, we can choose a *cost function*:

$$\mathcal{C}(\Theta) := \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}\|\boldsymbol{y}(\boldsymbol{x_i}) - \boldsymbol{a}^{[L]}(\boldsymbol{x_i})\|^2.$$

Here, Θ is the set of all parameters to optimise, namely $\Theta = \{W^{[2]}, \ldots, W^{[L]}, \boldsymbol{b}^{[2]}, \ldots, \boldsymbol{b}^{[L]}\}$ and the norm used is the Euclidean norm. Note that this dependence has been omitted on the right hand side, again for brevity. We now have a very clear goal, which is to optimise this cost function with respect to all parameters in Θ and to achieve this we will need to find a suitable optimisation algorithm. A common choice here would be *gradient descent*, a very well known method as outlined in Algorithm 1.

Algorithm 1 Gradient Descent

1: Start with a function F to optimise with respect to $\boldsymbol{\theta}$
2: Choose an initial point $\boldsymbol{\theta}_0$ and a step size η
3: **for** $k = 1, 2, \ldots$ until convergence **do**
4: $\quad \boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta\nabla F(\boldsymbol{\theta}_k)$
5: **end for**

This is only a brief formulation of the algorithm, it has obvious improvements like stopping criterion etc. but we will keep it simple here. The parameter η is called the step size, although in a machine learning context is usually known as the *learning rate*. The choice of this is important, as a larger choice results in faster learning but a value too large could result in jumping over minima and eventual divergence. Applied to our context, calculating

the gradient of $\mathcal{C}$ can be approached term by term. Let's define

$$C_i(\Theta) := \frac{1}{2}\|\boldsymbol{y}(\boldsymbol{x_i}) - \boldsymbol{a}^{[L]}(\boldsymbol{x_i})\|^2 = \frac{1}{2}\sum_{m=1}^{n_L}\left(\boldsymbol{y}(\boldsymbol{x_i})_m - \boldsymbol{a}^{[L]}(\boldsymbol{x_i})_m\right)^2 \tag{4}$$

which allows us to express the gradient as the summation

$$\nabla\mathcal{C}(\Theta) = \frac{1}{N}\sum_{i=1}^{N}\nabla C_i(\Theta).$$

Prohibitively, if N is very large as would be expected in practice, this sum is going to be expensive to compute for every iteration of gradient descent. Instead we can consider something called *stochastic gradient descent*. This is a variation on gradient descent that specifically applies to summations, as we have here. The idea is that, instead of averaging the gradient over all training points, we choose a single point to represent the entire function. Although this seems ineffective, it actually turns out to work incredibly well and further variations on this concept can make this method even more accurate. Here we will consider an algorithm that randomly samples without replacement, but there are countless variations that might work better with certain problems. The pseudocode for this (in our context) is outlined in Algorithm 2.

Algorithm 2 Stochastic Gradient Descent

```
Choose an initial set of parameters Θ and a learning rate η
repeat
    Permute {1, ..., N} to obtain a new order {k_1, ..., k_N}
    for k = k_1, ..., k_N do
        for each parameter θ ∈ Θ do
            θ = θ − η∇C_k(θ)
        end for
    end for
until convergence
```

Unfortunately, each iteration is not guaranteed to reduce the cost function as we are not using the true gradient for the updates. However, in practice we can achieve a good minimum candidate much faster than with using the standard gradient descent method.

2.4 Back Propagation

We now have an algorithm to solve our problem, so the only things we are missing are the derivatives of C_i that the algorithm requires. Before we can compute these we need to define the *Hadamard Product*, denoted by $\circ$, which is a essentially element-wise multiplication of vectors. Formally,

$$(\boldsymbol{v} \circ \boldsymbol{w})_j = v_j w_j.$$

Now we can look to Lemma 2.1 which computes the required derivatives. To make this more digestible we will use the notation in (1) - (3) but drop the dependence on $\boldsymbol{x}$ (consequently dropping the subscript on C_i).

Lemma 2.1. *Choosing σ as our activation function and defining $\boldsymbol{\delta}^{[l]}$ component-wise as*

$$\delta_j^{[l]} := \frac{\partial C}{\partial z_j^{[l]}}$$

we can show that,

$$\boldsymbol{\delta}^{[L]} = \boldsymbol{\sigma}'(\boldsymbol{z}^{[L]}) \circ (\boldsymbol{a}^{[L]} - \boldsymbol{y}) \tag{5}$$

$$\boldsymbol{\delta}^{[l]} = \boldsymbol{\sigma}'(\boldsymbol{z}^{[l]}) \circ (W^{[l+1]})^T \boldsymbol{\delta}^{[l+1]} \qquad \text{for } 2 \leq l \leq L-1 \tag{6}$$

$$\frac{\partial C}{\partial b_j^{[l]}} = \delta_j^{[l]} \qquad \text{for } 2 \leq l \leq L \tag{7}$$

$$\frac{\partial C}{\partial w_{jk}^{[l]}} = \delta_j^{[l]} a_k^{[l-1]} \qquad \text{for } 2 \leq l \leq L. \tag{8}$$

Proof. For these proofs it is easiest to prove the component-wise forms of the above. Firstly lets consider the proof of (5). By the chain rule we have that

$$\delta_j^{[L]} = \frac{\partial C}{\partial z_j^{[L]}} = \frac{\partial C}{\partial a_j^{[L]}} \frac{\partial a_j^{[L]}}{\partial z_j^{[L]}}.$$

As $\boldsymbol{a}^{[l]} = \sigma(\boldsymbol{z}^{[l]})$ for $2 \leq l \leq L$ by (1),

$$\frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} = \sigma'(z_j^{[L]}).$$

Also from the definition of C in (4),

$$\frac{\partial C}{\partial a_j^{[L]}} = \frac{\partial}{\partial a_j^{[L]}} \frac{1}{2} \sum_{m=1}^{n_L} (y_m - a_m^{[L]})^2 = a_j^{[L]} - y_j.$$

Therefore,

$$\delta_j^{[L]} = \sigma'(z_j^{[L]})(a_j^{[L]} - y_j)$$

as required. Now lets consider (6). We can start by using the chain rule for partial differentiation to obtain

$$\delta_j^{[l]} = \frac{\partial C}{\partial z_j^{[l]}} = \sum_{m=1}^{n_{l+1}} \frac{\partial C}{\partial z_m^{[l+1]}} \frac{\partial z_m^{[l+1]}}{\partial z_j^{[l]}} = \sum_{m=1}^{n_{l+1}} \delta_m^{[l+1]} \frac{\partial z_m^{[l+1]}}{\partial z_j^{[l]}}.$$

Then by the definition of $\boldsymbol{z}^{[l]}$ (2) we have,

$$\frac{\partial z_m^{[l+1]}}{\partial z_j^{[l]}} = \frac{\partial}{\partial z_j^{[l]}} \left(\sum_{p=1}^{n_l} \left[w_{mp}^{[l+1]} \sigma(z_p^{[l]}) \right] + b_m^{[l+1]} \right) = w_{mj}^{[l+1]} \sigma'(z_j^{[l]})$$

$$\implies \delta_j^{[l]} = \sum_{m=1}^{n_{l+1}} \delta_m^{[l+1]} w_{mj}^{[l+1]} \sigma'(z_j^{[l]}) = \sigma'(z_j^{[l]}) \left((W^{[l+1]})^T \boldsymbol{\delta}^{[l+1]} \right)_j$$

as required. To show (7) first note that

$$z_j^{[l]} = \left(W^{[l]}\boldsymbol{\sigma}(\boldsymbol{z}^{[l-1]})\right)_j + b_j^{[l]} \tag{9}$$
$$\implies \frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = 1$$

so using the chain rule we can simplify

$$\frac{\partial C}{\partial b_j^{[l]}} = \frac{\partial C}{\partial z_j^{[l]}}\frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = \delta_j^{[l]} \cdot 1 = \delta_j^{[l]}$$

as required. Finally to prove (8) we can use an expanded version of (9),

$$\frac{\partial z_m^{[l]}}{\partial w_{jk}^{[l]}} = \frac{\partial}{\partial w_{jk}^{[l]}}\left(\sum_{p=1}^{n_{l-1}}\left[w_{mp}^{[l]}a_p^{[l-1]}\right] + b_m^{[l]}\right) = \begin{cases} a_k^{[l-1]} & \text{if } m = j \\ 0 & \text{otherwise} \end{cases}.$$

Then using this and again the chain rule for partial differentiation, we obtain

$$\frac{\partial C}{\partial w_{jk}^{[l]}} = \sum_{m=1}^{n_l}\frac{\partial C}{\partial z_m^{[l]}}\frac{\partial z_m^{[l]}}{\partial w_{jk}^{[l]}} = \frac{\partial C}{\partial z_j^{[l]}}\frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}} = \delta_j^{[l]}a_k^{[l-1]}.$$

This completes the proof. □

This result almost gives us everything we need to start training the network, but not quite in the required form. We can look to Corollary 2.1 to give us the completed results.

Corollary 2.1. *If we define*

$$\left(\nabla_{\boldsymbol{b}}^{[l]}C\right)_j := \frac{\partial C}{\partial b_j^{[l]}} \qquad \left(\nabla_W^{[l]}C\right)_{jk} := \frac{\partial C}{\partial w_{jk}^{[l]}}$$

and use $\mathbf{1}$ *to represent a vector of 1's, we have that,*

$$\boldsymbol{\sigma}'(\boldsymbol{z}) = \boldsymbol{\sigma}(\boldsymbol{z}) \circ (\mathbf{1} - \boldsymbol{\sigma}(\boldsymbol{z})) \tag{10}$$
$$\boldsymbol{\delta}^{[L]} = \boldsymbol{a}^{[L]} \circ (\mathbf{1} - \boldsymbol{a}^{[L]}) \circ (\boldsymbol{a}^{[L]} - \boldsymbol{y}) \tag{11}$$
$$\boldsymbol{\delta}^{[l]} = \boldsymbol{a}^{[l]} \circ (\mathbf{1} - \boldsymbol{a}^{[l]}) \circ (W^{[l+1]})^T\boldsymbol{\delta}^{[l+1]} \qquad \text{for } 2 \le l \le L-1 \tag{12}$$
$$\nabla_{\boldsymbol{b}}^{[l]}C = \boldsymbol{\delta}^{[l]} \qquad \text{for } 2 \le l \le L \tag{13}$$
$$\nabla_W^{[l]}C = \boldsymbol{\delta}^{[l]}(\boldsymbol{a}^{[l-1]})^T \qquad \text{for } 2 \le l \le L. \tag{14}$$

Proof. To prove (10) we can compute the derivative directly:

$$\begin{aligned}\sigma'(z) &= \frac{d}{dz}(1+e^{-z})^{-1} = e^{-z}(1+e^{-z})^{-2} = (1+e^{-z}-1)(1+e^{-z})^{-2} \\ &= (1+e^{-z})^{-1} - (1+e^{-z})^{-2} = (1+e^{-z})^{-1}(1-(1+e^{-z})^{-1}) \\ &= \sigma(z)(1-\sigma(z))\end{aligned}$$

which is the component-wise form of (10) as required. Now using this result and (2) we have that,

$$\boldsymbol{\sigma}'(\boldsymbol{z}^{[L]}) = \boldsymbol{\sigma}(\boldsymbol{z}^{[L]}) \circ (\mathbf{1} - \boldsymbol{\sigma}(\boldsymbol{z}^{[L]})) = \boldsymbol{a}^{[L]} \circ (\mathbf{1} - \boldsymbol{a}^{[L]})$$

which from (5) and (6) in Lemma 2.1 immediately proves (11) and (12). Furthermore, (13) follows directly from Lemma 2.1, which has the result but in component-wise form (7). For the last result (14) we can notice that for any two vectors $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^n$,

$$(\boldsymbol{u}\boldsymbol{v}^T)_{jk} = u_j v_k$$

so applying this to (8) in Lemma 2.1 completes the proof. □

Now if you inspect this result you will notice that in order to calculate all of these partial derivatives you must start from $\boldsymbol{\delta}^{[L]}$ and work backwards. Furthermore to do this you need $\boldsymbol{a}^{[L]}$, but this can easily be found by running a forward pass through the network and computing the final output. This method of calculating gradients is a special case of automatic differentiation known as *back propagation* and is very widely used to train neural networks.

With all of this information we can now formulate a full algorithm using stochastic gradient descent to train a neural network from scratch. The pseudocode for this is in Algorithm 3. Here we have mentioned the term *training epoch*, or simply epoch, which is a single pass through all training examples. Essentially when the algorithm has seen all the training data once, it has completed an epoch. Also notice that the weights and biases are initialised randomly as this helps prevent a tendency towards a particular output.

To see what this algorithm actually does, we can look a classification example. For simplicity we will use two-dimensional inputs with two output classes (demonstrated as red and blue). After training a neural network using Algorithm 3 we get the result in Figure 3 and you can see that the majority of the points have been correctly classified. The advantages of using something such as the sigmoid function for activation include its ability to account for uncertainty in results, as illustrated by the shading in-between red and blue in Figure 3a. These areas indicate that the outputs were not strong enough to place the points in either class. The value of the cost function for this problem is plotted in Figure 3b and you can see that it is far from smooth, which is a consequence of using stochastic gradient descent and therefore introducing the random peaks.

We can also investigate the same example but with more layers and a higher quantity of neurons in each layer in Figure 4. Although from Figure 4a it seems that the network would perform better than the one considered in Figure 3, this might not be the case. If we look at the cost function for the second example in Figure 4b, while the cost for training data is decreasing past 800 epochs; the cost for the validation data is actually increasing. This could be an example of overfitting, where the algorithm is starting to learn too much of the training data for it to provide reliable results on unseen data.

2.5 Other Things to Consider

There are many improvements that can be made on the simple design formulated in Algorithm 3 and although we won't be detailing those here, there is one seen in almost all training algorithms that is worth mentioning. This variation uses a modified stochastic gradient descent algorithm where the gradient is calculated in mini batches. So instead of

Algorithm 3 Training a Neural Network

Start with the training data $(\boldsymbol{x}_i)_{i=1}^N$, $(\boldsymbol{y}_i)_{i=1}^N$; choose a learning rate η and a number of training epochs E

Initialise $W^{[2]}, \ldots, W^{[L]}, \boldsymbol{b}^{[2]}, \ldots, \boldsymbol{b}^{[L]}$ randomly
for $i = 1, \ldots E$ **do**

    Permute $\{1, \ldots, N\}$ to obtain a new order $\{k_1, \ldots, k_N\}$

    for $k = k_1, \ldots, k_N$ **do**

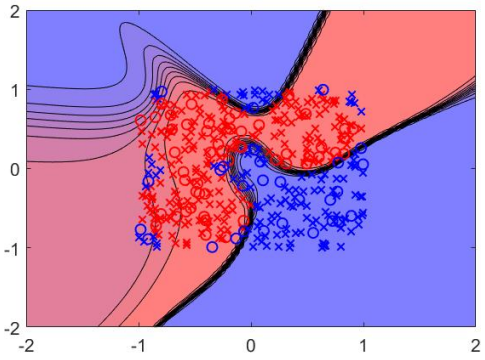
        $\boldsymbol{a}^{[1]} = \boldsymbol{x}$
        for $j = 2, \ldots, L$ **do**
            $\boldsymbol{a}^{[j]} = g^{[j]}(\boldsymbol{a}^{[j-1]})$
        end for

        $\boldsymbol{\delta}^{[L]} = \boldsymbol{a}^{[L]} \circ (\boldsymbol{1} - \boldsymbol{a}^{[L]}) \circ (\boldsymbol{a}^{[L]} - \boldsymbol{y})$
        for $j = 2, \ldots, L-1$ **do**
            $\boldsymbol{\delta}^{[j]} = \boldsymbol{a}^{[j]} \circ (\boldsymbol{1} - \boldsymbol{a}^{[j]}) \circ (W^{[j+1]})^T \boldsymbol{\delta}^{[j+1]}$
        end for

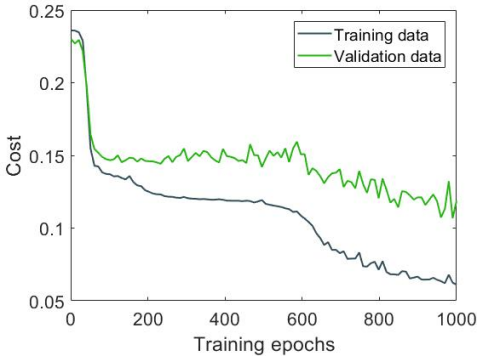
        for $j = 2, \ldots, L$ **do**
            $\boldsymbol{b}^{[j]} = \boldsymbol{b}^{[j]} - \eta \boldsymbol{\delta}^{[j]}$
            $W^{[j]} = W^{[j]} - \eta \boldsymbol{\delta}^{[j]} (\boldsymbol{a}^{[j-1]})^T$
        end for

    end for
end for

return $W^{[2]}, \ldots, W^{[L]}, \boldsymbol{b}^{[2]}, \ldots, \boldsymbol{b}^{[L]}$

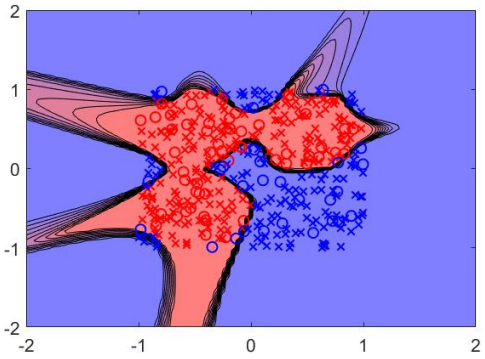


(a) Neural network classification. Circles indicate test data and crosses training data.

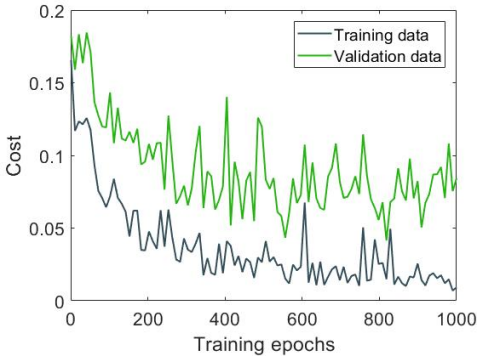


(b) The value of the cost function at each training epoch.

Figure 3: Training a neural network with 6 layers; a hidden neuron pattern of 6, 8, 9, 7; 1000 training epochs and a learning rate of 0.05.



(a) Neural network classification. Circles indicate test data and crosses training data.



(b) The value of the cost function at each training epoch.

Figure 4: Training a neural network with 12 layers; a hidden neuron pattern of 60, 82, 95, 70, 44, 23, 50, 45, 44, 72; 1000 training epochs and a learning rate of 0.05.

computing the gradient at each input point, we partition the set of input points and compute averages of their gradients for use in each iteration which results in much more stable training and better results [5].

Another thing to consider is how to measure the performance of a trained neural network. This becomes particularly important when trying to solve Problem 2 and we clearly need some metric to compare networks with different architectures and starting parameters. We could use something such as the value of the cost function, but there is a more common approach. This involves splitting the input data into training data and test/validation data. Then only the training data is given to the network to learn from and after training is complete we see how many test points were accurately predicted. This then gives us another metric to compare different networks, let's call it the *validation accuracy*, which is the percentage of validation points accurately classified.

3 Optimising Hyperparameters

So now we have managed to look at how to train a neural network in detail we can move onto Problem 2 which is choosing parameters of the network. In Section 2 we have seen a few parameters that would be useful to optimise (the architecture etc.) although there are many variations on the algorithm that would introduce more. From now on we will refer to these parameters as *hyperparameters* which distinguishes them from the those determined as a result of training.

Let's introduce an objective function f that outputs a measure of performance of the network based upon input of all the hyperparameters we seek to optimise. As mentioned in Section 1.1 f is a very complex function to optimise in traditional ways, so we can consider it as a black box function. If it weren't for the time it took to evaluate f then we could automate the choice of hypereparameters by means of a grid search, but this would clearly be infeasible for even a single hyperparameter. Instead suppose we could perform a similar sort of search intelligently and only search in areas that are most likely to yield useful results. In this section we will explore a solution using Bayesian statistics, where we model f as a random variable and generate predictions based upon previous observations of the training process. Even though this approach will not guarantee it will find an optimum, it will be computationally feasible.

For simplicity, we will fix the network performance metric as its validation accuracy as outlined in Section 2.5 although there are many other good choices, none of which would affect our investigation. Confusingly the problem we are trying to solve is very much a meta problem, predicting output based on multidimensional input which is almost an identical problem that the neural networks solve in the first place. So from now on, we will define the input $\boldsymbol{x}$ to mean the vector of hyperparameters (not the training data) and the output y to be an observation of the validation accuracy of the network. Unless explicitly written otherwise, subsequent notation can be considered as being disconnected from all previous sections.

3.1 Bayesian Optimisation

Bayesian optimisation is a strategy to optimise black box functions [8] [9]. The basic concept is to take a black box objective function f, model it as a random variable, then search for an optimum based upon statistics established from previous observations. This strategy is known as sequential, which means that it does not use a predetermined sample size and instead data is analysed as it is sampled. To better comprehend this strategy we can formulate a summary:

1. Start with an objective function f to optimise over $\boldsymbol{x}$, and an acquisition function a
2. Model f as a Gaussian process
3. Find a point to sample by optimising a given all previous observations
4. Obtain a sample at this point and add it to the set of observations
5. Update the Gaussian process posterior
6. Repeat steps 3 to 6.

To be able to implement this we can start by modelling samples from f as random variables and formulate a conditional posterior distribution based upon previous observations. We can then extend this model to a Gaussian process before finally choosing a suitable acquisition function that decides which point to sample at each iteration. The introductory material for this chapter was taken from Rasmussen and Williams [7] but was extended to include proofs for important results, a more in-depth look at basis functions and a rigorous analysis of sampling error.

Before continuing it would be useful to define a multivariate normal distribution, which is a generalisation of the univariate normal distribution and will be used frequently in the following sections.

Definition 3.1. *A vector $\boldsymbol{X} \in \mathbb{R}^k$ follows a k-variate (multivariate) normal distribution if every linear combination of its components has a univariate normal distribution. Formally,*

$$\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma) \iff \sum_{i=1}^{k} \gamma_i X_i \sim \mathcal{N}(\mu_i, \Sigma_{ii}) \quad \forall\, \gamma_i \in \mathbb{R},\, i \in \{1, \dots, k\}$$

with k-dimensional mean vector and $k \times k$ covariance matrix,

$$\begin{aligned} \boldsymbol{\mu} &= [\mathbb{E}(X_1), \dots, \mathbb{E}(X_k)]^T \\ \Sigma &= [\mathrm{Cov}(X_i, X_j); 1 \le i, j \le k]. \end{aligned}$$

Its probability density function is

$$f_{\boldsymbol{X}}(\boldsymbol{x}) = (2\pi)^{-k/2} \det(\Sigma)^{-1/2} \exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu})^T \Sigma^{-1} (\boldsymbol{x} - \boldsymbol{\mu})\right).$$

3.2 Conditional Distributions

Now our approach to implementing Bayesian optimisation starts with statistically modelling samples of f at untested points given previous samples. For this we can prove a very powerful result on conditional distributions.

Theorem 3.1. *Let $\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$ such that,*

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{X_1} \\ \boldsymbol{X_2} \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \boldsymbol{\mu_1} \\ \boldsymbol{\mu_2} \end{bmatrix}, \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix} \right).$$

Then we have that,

$$\boldsymbol{X_2} \mid \boldsymbol{X_1} = \boldsymbol{x_1} \sim \mathcal{N}\left(\boldsymbol{\mu_2} + \Sigma_{12}^T \Sigma_{11}^{-1} [\boldsymbol{x_1} - \boldsymbol{\mu_1}],\, \Sigma_{22} - \Sigma_{12}^T \Sigma_{11}^{-1} \Sigma_{12}\right).$$

Proof. First we can use the definition of the pdf of a conditional distribution,

$$f_{\boldsymbol{X_2} \mid \boldsymbol{X_1}}(\boldsymbol{x} \mid \boldsymbol{x_1}) = \frac{f_{\boldsymbol{X_1}, \boldsymbol{X_2}}(\boldsymbol{x_1}, \boldsymbol{x})}{f_{\boldsymbol{X_1}}(\boldsymbol{x_1})}. \tag{15}$$

Then considering only the numerator (the joint distribution of $\boldsymbol{X_1}$ and $\boldsymbol{X_2}$) we get:

$$\begin{aligned} f_{\boldsymbol{X_1}, \boldsymbol{X_2}}(\boldsymbol{x_1}, \boldsymbol{x}) &= \frac{1}{\sqrt{(2\pi)^n |\Sigma|}} \exp\left(-\frac{1}{2} \begin{bmatrix} \boldsymbol{x_1} - \boldsymbol{\mu_1} \\ \boldsymbol{x} - \boldsymbol{\mu_2} \end{bmatrix}^T \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}^{-1} \begin{bmatrix} \boldsymbol{x_1} - \boldsymbol{\mu_1} \\ \boldsymbol{x} - \boldsymbol{\mu_2} \end{bmatrix}\right) \\ &= \frac{1}{\sqrt{(2\pi)^n |\Sigma|}} \exp\left(-\frac{1}{2} Q(\boldsymbol{x_1}, \boldsymbol{x})\right) \end{aligned} \tag{16}$$

where we define

$$Q(\boldsymbol{x_1}, \boldsymbol{x}) := \begin{bmatrix} \boldsymbol{x_1} - \boldsymbol{\mu_1} \\ \boldsymbol{x} - \boldsymbol{\mu_2} \end{bmatrix}^T \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}^{-1} \begin{bmatrix} \boldsymbol{x_1} - \boldsymbol{\mu_1} \\ \boldsymbol{x} - \boldsymbol{\mu_2} \end{bmatrix}.$$

Now to invert Σ in its block form we can use the Block Inversion Formula, which is straightforward to prove [3]. This gives us

$$\begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}^{-1} = \begin{bmatrix} \mathrm{T}_{11} & \mathrm{T}_{12} \\ \mathrm{T}_{21} & \mathrm{T}_{22} \end{bmatrix}$$

where the blocks Σ_{ij} and T_{ij} are of the same dimension. We now define

$$\Upsilon := \Sigma_{22} - \Sigma_{12}^T \Sigma_{11}^{-1} \Sigma_{12}.$$

Then if we assume from now on that Σ_{11} and $\Sigma_{22} - \Sigma_{12}^T \Sigma_{11}^{-1} \Sigma_{12}$ are non-singular then the Block Inversion Formula also gives us:

$$\begin{aligned} \mathrm{T}_{11} &= \Sigma_{11}^{-1} + \Sigma_{11}^{-1} \Sigma_{12} \Upsilon^{-1} \Sigma_{12}^T \Sigma_{11}^{-1} \\ \mathrm{T}_{12} &= -\Sigma_{11}^{-1} \Sigma_{12} \Upsilon^{-1} = \mathrm{T}_{21}^T \\ \mathrm{T}_{22} &= \Upsilon^{-1}. \end{aligned}$$

The second equality in which $\mathrm{T}_{12} = \mathrm{T}_{21}^T$ follows from the fact that (as Σ is a covariance matrix) Σ_{11}, Σ_{22} are symmetric and $\Sigma_{12} = \Sigma_{21}^T$. Noting that Υ is symmetric, we can substitute these in to get a complete expression for Q:

$$\begin{aligned} Q(\boldsymbol{x_1}, \boldsymbol{x}) &= [\boldsymbol{x_1} - \boldsymbol{\mu_1}]^T (\Sigma_{11}^{-1} + \Sigma_{11}^{-1} \Sigma_{12} \Upsilon^{-1} \Sigma_{12}^T \Sigma_{11}^{-1}) [\boldsymbol{x_1} - \boldsymbol{\mu_1}] \\ &\quad - 2[\boldsymbol{x_1} - \boldsymbol{\mu_1}]^T (\Sigma_{11}^{-1} \Sigma_{12} \Upsilon^{-1}) [\boldsymbol{x} - \boldsymbol{\mu_2}] + [\boldsymbol{x} - \boldsymbol{\mu_2}]^T \Upsilon^{-1} [\boldsymbol{x} - \boldsymbol{\mu_2}] \\ &= \left\{ [\boldsymbol{x_1} - \boldsymbol{\mu_1}]^T \Sigma_{11}^{-1} [\boldsymbol{x_1} - \boldsymbol{\mu_1}] \right\} + (\Sigma_{12}^T \Sigma_{11}^{-1} [\boldsymbol{x_1} - \boldsymbol{\mu_1}])^T \Upsilon^{-1} (\Sigma_{12}^T \Sigma_{11}^{-1} [\boldsymbol{x_1} - \boldsymbol{\mu_1}]) \\ &\quad - 2(\Sigma_{12}^T \Sigma_{11}^{-1} [\boldsymbol{x_1} - \boldsymbol{\mu_1}])^T \Upsilon^{-1} [\boldsymbol{x} - \boldsymbol{\mu_2}] + [\boldsymbol{x} - \boldsymbol{\mu_2}]^T \Upsilon^{-1} [\boldsymbol{x} - \boldsymbol{\mu_2}] \\ &= \left\{ [\boldsymbol{x_1} - \boldsymbol{\mu_1}]^T \Sigma_{11}^{-1} [\boldsymbol{x_1} - \boldsymbol{\mu_1}] \right\} + (\boldsymbol{x} - \boldsymbol{\lambda})^T \Upsilon^{-1} (\boldsymbol{x} - \boldsymbol{\lambda}) \end{aligned}$$

where $\boldsymbol{\lambda} := \boldsymbol{\mu_2} + \Sigma_{12}^T \Sigma_{11}^{-1} [\boldsymbol{x_1} - \boldsymbol{\mu_1}]$. Now if we substitute this back into (16) we obtain,

$$\begin{aligned} f_{\boldsymbol{X_1}, \boldsymbol{X_2}}(\boldsymbol{x_1}, \boldsymbol{x}) = \frac{1}{\sqrt{(2\pi)^n |\Sigma|}} &\exp\left(-\frac{1}{2} [\boldsymbol{x_1} - \boldsymbol{\mu_1}]^T \Sigma_{11}^{-1} [\boldsymbol{x_1} - \boldsymbol{\mu_1}] \right) \\ &\exp\left(-\frac{1}{2} (\boldsymbol{x} - \boldsymbol{\lambda})^T \Upsilon^{-1} (\boldsymbol{x} - \boldsymbol{\lambda}) \right). \end{aligned}$$

It can then be proved with some more linear algebra [10] that

$$|\Sigma| = \begin{vmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{vmatrix} = |\Sigma_{11}||\Upsilon|$$

and if we say that $\boldsymbol{X_1} \in \mathbb{R}^{n_1}$, $\boldsymbol{X_2} \in \mathbb{R}^{n_2}$ we have,

$$\begin{aligned} f_{\boldsymbol{X_1}, \boldsymbol{X_2}}(\boldsymbol{x_1}, \boldsymbol{x}) &= \frac{1}{\sqrt{(2\pi)^{n_1} |\Sigma_{11}|}} \exp\left(-\frac{1}{2} [\boldsymbol{x_1} - \boldsymbol{\mu_1}]^T \Sigma_{11}^{-1} [\boldsymbol{x_1} - \boldsymbol{\mu_1}] \right) \\ &\quad \frac{1}{\sqrt{(2\pi)^{n_2} |\Upsilon|}} \exp\left(-\frac{1}{2} (\boldsymbol{x} - \boldsymbol{\lambda})^T \Upsilon^{-1} (\boldsymbol{x} - \boldsymbol{\lambda}) \right) \\ &= f_{\boldsymbol{X_1}}(\boldsymbol{x_1}) \; g(\boldsymbol{x_1}, \boldsymbol{x}). \end{aligned}$$

Finally substituting this back into (15) we can see that,

$$f_{\boldsymbol{X_2}|\boldsymbol{X_1}}(\boldsymbol{x} \mid \boldsymbol{x_1}) = g(\boldsymbol{x_1}, \boldsymbol{x})$$
$$\implies \boldsymbol{X_2} \mid \boldsymbol{X_1} = \boldsymbol{x_1} \sim \mathcal{N}\left(\boldsymbol{\mu_2} + \Sigma_{12}^T \Sigma_{11}^{-1}[\boldsymbol{x_1} - \boldsymbol{\mu_1}],\, \Sigma_{22} - \Sigma_{12}^T \Sigma_{11}^{-1} \Sigma_{12}\right)$$

which is the required result. □

So providing we have a random vector $\boldsymbol{X}$ of untested and tested/observed values, we can build a conditional distribution of the untested points based upon previous observations.

3.3 Linear Regression

Let's consider a simplified version of our function f which has single-dimensional input. We can then propose the simple linear model

$$f(x) = wx. \tag{17}$$

Here w can be considered as a weight, and we can place a prior such that $w \sim \mathcal{N}\left(0,\, \alpha^2\right)$ for some $\alpha \in \mathbb{R}$. Now using this to model actual observed values is somewhat idealistic, as in reality we only have access to noisy versions of them. In our context training a neural network introduces the noise through the random initialisation before training. So to account for this, we can model observations as

$$y = f(x) + \epsilon \tag{18}$$

where ϵ is some sort of random effect such that $\epsilon \sim \mathcal{N}\left(0,\, \sigma^2\right)$ for $\sigma \in \mathbb{R}$. We now have two prior parameters to consider, α and σ but these have an intuitive purpose. The parameter σ represents how much error can be allowed when fitting the data so a low σ would result in a closer fit. Then α represents how steep the effect of the weight can be, thus a higher value of this parameter would increase the possible sensitivity a model has to a change in input.

Eventually we are trying to find a way to predict unseen values of $f(x)$ given some previous observations using Theorem 3.1. With this in mind we seek to find a distribution of a random variable $\boldsymbol{Y}$ given input points $\boldsymbol{x}$, both of which can be partitioned into tested and untested points. So let's denote $\boldsymbol{x_t}$, $\boldsymbol{Y_t}$ as the tested partitions and $\boldsymbol{x_u}$, $\boldsymbol{Y_u}$ as observed. Then we can write

$$\boldsymbol{Y} = \begin{bmatrix} \boldsymbol{Y_t} \\ \boldsymbol{Y_u} \end{bmatrix} = w \begin{bmatrix} \boldsymbol{x_t} \\ \boldsymbol{x_u} \end{bmatrix} + \begin{bmatrix} \boldsymbol{\epsilon} \\ \boldsymbol{0} \end{bmatrix} \quad \text{where } w \sim \mathcal{N}\left(0,\, \alpha^2\right),\ \boldsymbol{\epsilon} \sim \mathcal{N}\left(0,\, \sigma^2 I\right).$$

Thus as w and $\boldsymbol{\epsilon}$ are normal (and independent) we can model their joint distribution:

$$\begin{bmatrix} w \\ \boldsymbol{\epsilon} \end{bmatrix} \sim \mathcal{N}\left(\boldsymbol{0},\, \begin{bmatrix} \alpha^2 & 0 \\ 0 & \sigma^2 I \end{bmatrix}\right). \tag{19}$$

Our aim however is to formulate a distribution of $\boldsymbol{Y}$. It turns out that we can rewrite $\boldsymbol{Y}$ as a linear transformation of (19), so

$$\boldsymbol{Y} = \begin{bmatrix} \boldsymbol{x} & I \\ & 0 \end{bmatrix} \begin{bmatrix} w \\ \boldsymbol{\epsilon} \end{bmatrix} \qquad I \in \mathbb{R}^{n \times n}$$

where n is the number of observed points. This allows us to find a distribution for $\boldsymbol{Y}$ with a well known result on affine transformations of multivariate normal distributions.

Theorem 3.2. *Let* $\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$ *and* $\boldsymbol{Y} = A\boldsymbol{X}+\boldsymbol{b}$ *for* $A \in \mathbb{R}^{N\times N}$ *(invertible);* $\boldsymbol{X}, \boldsymbol{b} \in \mathbb{R}^N$. *Then,*

$$\boldsymbol{Y} \sim \mathcal{N}\left(A\boldsymbol{\mu} + \boldsymbol{b},\, A\Sigma A^T\right).$$

Proof. This can be proved using a change of variables result [11] for multivariate probability distributions which states: for $\boldsymbol{Y} = H(\boldsymbol{X})$ where H is differentiable and bijective,

$$f_{\boldsymbol{Y}}(\boldsymbol{y}) = \left|\det(D(H^{-1}))\right| f_{\boldsymbol{X}}(H^{-1}(\boldsymbol{y})) \tag{20}$$

where f represents a pdf and $D(H^{-1})$ represents the Jacobian of H^{-1}. In our case we have that $H(\boldsymbol{X}) = A\boldsymbol{X} + \boldsymbol{b}$ which is clearly differentiable. Then the bijection is trivial to prove for an invertible A. First,

$$\begin{aligned} H(\boldsymbol{U}) = H(\boldsymbol{V}) \quad &\text{for some } \boldsymbol{U}, \boldsymbol{V} \in \mathbb{R}^N \\ &\iff A\boldsymbol{U} + \boldsymbol{b} = A\boldsymbol{V} + \boldsymbol{b} \\ &\iff \boldsymbol{U} = \boldsymbol{V} \end{aligned}$$

so H is well defined and injective. Then for surjectivity let $\boldsymbol{U} \in \mathbb{R}^N$. Then $\boldsymbol{U} = H(A^{-1}(\boldsymbol{U} - \boldsymbol{b}))$ so H is surjective and therefore bijective. Consequently we can find the pdf of $\boldsymbol{Y}$ using (20) as follows:

$$\begin{aligned} f_{\boldsymbol{Y}}(\boldsymbol{y}) &= \left|\det(A^{-1})\right| \frac{1}{\sqrt{(2\pi)^n|\Sigma|}} \exp\left(-\frac{1}{2}(A^{-1}(\boldsymbol{y}-\boldsymbol{b}) - \boldsymbol{\mu})^T\Sigma^{-1}(A^{-1}(\boldsymbol{y}-\boldsymbol{b}) - \boldsymbol{\mu})\right) \\ &= \frac{1}{\sqrt{(2\pi)^n|\Sigma||A|^2}} \exp\left(-\frac{1}{2}Q(\boldsymbol{y})\right) \end{aligned}$$

where we have used that $|A^{-1}| = |A|^{-1}$ and,

$$\begin{aligned} Q(\boldsymbol{y}) :&= (\boldsymbol{y}-\boldsymbol{b})^T A^{-T}\Sigma^{-1}A^{-1}(\boldsymbol{y}-\boldsymbol{b}) - (\boldsymbol{y}-\boldsymbol{b})^T A^{-T}\Sigma^{-1}\boldsymbol{\mu} \\ &\quad - \boldsymbol{\mu}^T\Sigma^{-1}A^{-1}(\boldsymbol{y}-\boldsymbol{b}) + \boldsymbol{\mu}^T\Sigma^{-1}\boldsymbol{\mu} \\ &= (\boldsymbol{y}-\boldsymbol{b})^T(A\Sigma A^T)^{-1}(\boldsymbol{y}-\boldsymbol{b}) - (\boldsymbol{y}-\boldsymbol{b})^T(A\Sigma A^T)^{-1}(A\boldsymbol{\mu}) \\ &\quad - (A\boldsymbol{\mu})^T(A\Sigma A^T)^{-1}(\boldsymbol{y}-\boldsymbol{b}) + (A\boldsymbol{\mu})^T(A\Sigma A^T)^{-1}(A\boldsymbol{\mu}) \\ &= ((\boldsymbol{y}-\boldsymbol{b}) - A\boldsymbol{\mu})^T(A\Sigma A^T)^{-1}((\boldsymbol{y}-\boldsymbol{b}) - A\boldsymbol{\mu}) \\ &= (\boldsymbol{y} - (A\boldsymbol{\mu}+\boldsymbol{b}))^T(A\Sigma A^T)^{-1}(\boldsymbol{y} - (A\boldsymbol{\mu}+\boldsymbol{b})). \end{aligned}$$

We can also notice that, as $|A| = |A^T|$ and the determinant is multiplicative for square matrices,

$$|\Sigma||A|^2 = |A||\Sigma||A^T| = |A\Sigma A^T|.$$

So finally we have that

$$\begin{aligned} f_{\boldsymbol{Y}}(\boldsymbol{y}) &= \frac{1}{\sqrt{(2\pi)^n|A\Sigma A^T|}} \exp\left(-\frac{1}{2}(\boldsymbol{y} - (A\boldsymbol{\mu}+\boldsymbol{b}))^T(A\Sigma A^T)^{-1}(\boldsymbol{y} - (A\boldsymbol{\mu}+\boldsymbol{b}))\right) \\ &\implies \boldsymbol{Y} \sim \mathcal{N}\left(A\boldsymbol{\mu} + \boldsymbol{b},\, A\Sigma A^T\right) \end{aligned}$$

as required. □

The distribution of $\boldsymbol{Y}$ then follows immediately as a result of Theorem 3.2. So,

$$\boldsymbol{Y} = \begin{bmatrix} \boldsymbol{x} & I \\ & 0 \end{bmatrix} \begin{bmatrix} w \\ \boldsymbol{\epsilon} \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0},\, \Sigma_{\boldsymbol{Y}}\right) \tag{21}$$

where formulation of the covariance matrix $\Sigma_{\boldsymbol{Y}}$ follows from some simple linear algebra:

$$\Sigma_{\boldsymbol{Y}} = \begin{bmatrix} \boldsymbol{x} & I \\ & 0 \end{bmatrix} \begin{bmatrix} \alpha^2 & 0 \\ 0 & \sigma^2 I \end{bmatrix} \begin{bmatrix} \boldsymbol{x}^T & \\ I & 0 \end{bmatrix} = \begin{bmatrix} \alpha^2 \boldsymbol{x_t} \boldsymbol{x_t}^T + \sigma^2 I & \alpha^2 \boldsymbol{x_t} \boldsymbol{x_u}^T \\ \alpha^2 \boldsymbol{x_u} \boldsymbol{x_t}^T & \alpha^2 \boldsymbol{x_u} \boldsymbol{x_u}^T \end{bmatrix}.$$

Hence using Theorem 3.1 we can finally derive the conditional distribution to be

$$\begin{aligned} \boldsymbol{Y_u} \mid \boldsymbol{Y_t} = \boldsymbol{y_t} &\sim \mathcal{N}\left(\boldsymbol{\mu},\, \Sigma\right) \\ \boldsymbol{\mu} &= \alpha^2 \boldsymbol{x_u} \boldsymbol{x_t}^T (\alpha^2 \boldsymbol{x_t} \boldsymbol{x_t}^T + \sigma^2 I)^{-1} \boldsymbol{y_t} \\ \Sigma &= \alpha^2 \boldsymbol{x_u} \boldsymbol{x_u}^T - \alpha^4 \boldsymbol{x_u} \boldsymbol{x_t}^T (\alpha^2 \boldsymbol{x_t} \boldsymbol{x_t}^T + \sigma^2 I)^{-1} \boldsymbol{x_t} \boldsymbol{x_u}^T. \end{aligned}$$

Now at the moment we are working with the restriction of a one-dimensional input whereas this is rarely the case in practice. Fortunately extending all of our previous analysis to work in higher dimensions is straightforward and turns out to yield intuitively similar results.

3.4 Extending to Higher Dimensions

First we can consider a similar model to before in (17), but now we can consider a multidimensional input, and therefore multiple weights:

$$\begin{gathered} f(\boldsymbol{x}) = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_d x_d \qquad y = f(\boldsymbol{x}) + \epsilon \\ \epsilon \sim \mathcal{N}\left(0,\, \sigma^2\right),\ w_j \sim \mathcal{N}\left(0,\, \alpha^2\right) \quad \forall\, j \in \{1, \cdots, d\}. \end{gathered} \tag{22}$$

Then consider data points $\boldsymbol{x}^{[1]}, \ldots, \boldsymbol{x}^{[n]}, \boldsymbol{x}^{[n+1]}, \ldots, \boldsymbol{x}^{[N]} \in \mathbb{R}^d$, with the first n being sampled points. Then we can write a system in a similar way to the one dimensional case:

$$\boldsymbol{Y} = \begin{bmatrix} 1 & x_1^{[1]} & x_2^{[1]} & \cdots & x_d^{[1]} \\ 1 & x_1^{[2]} & x_2^{[2]} & \cdots & x_d^{[2]} \\ & & \vdots & & \\ 1 & x_1^{[N]} & x_2^{[N]} & \cdots & x_d^{[N]} \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_d \end{bmatrix} + \begin{bmatrix} \boldsymbol{\epsilon} \\ \mathbf{0} \end{bmatrix} = X\boldsymbol{w} + \begin{bmatrix} \boldsymbol{\epsilon} \\ \mathbf{0} \end{bmatrix} \tag{23}$$

where $w_j \sim \mathcal{N}\left(0,\, \alpha^2\right),\ \epsilon_i \sim \mathcal{N}\left(0,\, \sigma^2\right) \quad \forall\, j \in \{1, \ldots, d\},\ i \in \{1, \ldots, n\}$.

In fact, we can generalise this further and instead of just restricting this analysis to take into account multiple dimensions we can also include transformations of our data points. So we define a function, let's call it a basis function, $\boldsymbol{\phi} : \mathbb{R}^d \to \mathbb{R}^q$ for some $q \in \mathbb{N}$. We can then extend the weights vector to be of length q such that $w_j \sim \mathcal{N}\left(0,\, \alpha^2\right)$ for $j \in \{1, \ldots, q\}$ and write a more general form of (23) as

$$\boldsymbol{Y} = \begin{bmatrix} \boldsymbol{\phi}(\boldsymbol{x}^{[1]})^T \\ \boldsymbol{\phi}(\boldsymbol{x}^{[2]})^T \\ \vdots \\ \boldsymbol{\phi}(\boldsymbol{x}^{[N]})^T \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_q \end{bmatrix} + \begin{bmatrix} \boldsymbol{\epsilon} \\ \mathbf{0} \end{bmatrix} = \boldsymbol{\phi}(X)\boldsymbol{w} + \begin{bmatrix} \boldsymbol{\epsilon} \\ \mathbf{0} \end{bmatrix} \tag{24}$$

where we use $\boldsymbol{\phi}(X)$ to represent applying $\boldsymbol{\phi}$ to each row in the matrix X. To see this basis function somewhat clearer, suppose we want to use quadratic regression as opposed to the linear approach in (22). We could then formulate $\boldsymbol{\phi}$ as

$$\boldsymbol{\phi} : \mathbb{R}^d \to \mathbb{R}^{2d+1}, \qquad \boldsymbol{x} \mapsto \begin{pmatrix} 1 & x_1 & \cdots & x_d & x_1^2 & \cdots & x_d^2 \end{pmatrix}^T.$$

Now we can formulate a joint distribution for $\boldsymbol{w}$ and $\boldsymbol{\epsilon}$ as in the single dimensional case previously and say that

$$\begin{bmatrix} \boldsymbol{w} \\ \boldsymbol{\epsilon} \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} \alpha^2 I & 0 \\ 0 & \sigma^2 I \end{bmatrix}\right).$$

From here we can do an almost identical observation as used to arrive at (21) (using Theorem 3.2). First we can partition the matrix $\boldsymbol{\phi}(X)$ as defined in (24):

$$\boldsymbol{\phi}(X) = \begin{bmatrix} \boldsymbol{\phi}(X_t) \\ \boldsymbol{\phi}(X_u) \end{bmatrix} = \left[\begin{array}{ccc|ccc} \boldsymbol{\phi}(\boldsymbol{x}^{[1]}) & \cdots & \boldsymbol{\phi}(\boldsymbol{x}^{[n]}) & \boldsymbol{\phi}(\boldsymbol{x}^{[n+1]}) & \cdots & \boldsymbol{\phi}(\boldsymbol{x}^{[N]}) \end{array}\right]^T.$$

This allows us to use Theorem 3.2 to formulate the distribution of $\boldsymbol{Y}$ as,

$$\boldsymbol{Y} = \boldsymbol{\phi}(X)\boldsymbol{w} + \begin{bmatrix} \boldsymbol{\epsilon} \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\phi}(X) & I \\ & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{w} \\ \boldsymbol{\epsilon} \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \Sigma_{\boldsymbol{Y}}\right) \tag{25}$$

where

$$\Sigma_{\boldsymbol{Y}} = \begin{bmatrix} \alpha^2 X_t X_t^T + \sigma^2 I & \alpha^2 X_t X_u^T \\ \alpha^2 X_u X_t^T & \alpha^2 X_u X_u^T \end{bmatrix}.$$

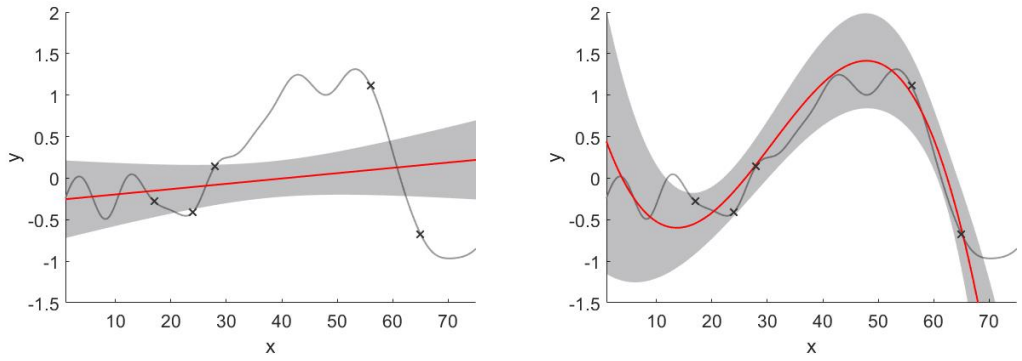
The logic in Section 3.3 can then be entirely reused by splitting $\boldsymbol{Y}$ into two partitions, $\boldsymbol{Y_t}$ and $\boldsymbol{Y_u}$, then using Theorem 3.1 again. This gives us the conditional distribution

$$\boldsymbol{Y_u} \mid \boldsymbol{Y_t} = \boldsymbol{y_t} \sim \mathcal{N}\left(\boldsymbol{\mu}, \Sigma\right)$$

where,

$$\begin{aligned} \boldsymbol{\mu} &= \alpha^2 X_u X_t^T (\alpha^2 X_t X_t^T + \sigma^2 I)^{-1} \boldsymbol{y_t} \\ \Sigma &= \alpha^2 X_u X_u^T - \alpha^4 X_u X_t^T (\alpha^2 X_t X_t^T + \sigma^2 I)^{-1} X_t X_u^T. \end{aligned}$$

It turns out our actual result is akin to the one dimensional version so with careful choice of the matrix $\boldsymbol{\phi}(X)$ (i.e. $\boldsymbol{\phi}$), the problem should not be any more difficult to deal with. Using this posterior, some plots were produced in Figure 5 that show regression using different basis functions. As you can see, the posterior in Figure 5b is a much more appropriate fit for the objective function than in Figure 5a. Unfortunately this is not very useful as a more unwieldy example may require even more complex basis functions. It does however illustrate the improvement that adding more complex basis functions gives us, so with this in mind we will explore a way to perform linear regression with an infinite number of basis functions in Section 3.5.



(a) Linear basis function.

(b) Cubic basis function.

Figure 5: Posterior drawn from the linear regression model ($\alpha = 1$, $\sigma = 0.25$) with different basis functions. Here the black line shows the objective function; the crosses are sampled points; the red line indicates the posterior mean whilst the shaded region is the mean ± 1.96 times the posterior standard deviation.

3.5 Gaussian Processes

By introducing this basis function $\boldsymbol{\phi}$ we have allowed boundless improvement on the original linear model. However the process of choosing this function is not only fully problem dependant but it can be viewed as a choice of a complex hyperparameter which is exactly what we are trying to avoid. We can work towards solving this problem by modelling f as a Gaussian process [7] which we will define here.

Definition 3.2. *A Gaussian process is a collection of random variables, such that any finite number of them follow a multivariate normal distribution. This process is completely specified by a mean and covariance function.*

The covariance function mentioned is also known as the *kernel*, which is the terminology we will use going forward. We consider modelling the real process f as a Gaussian process and we can specify a logical choice of mean and covariance function as follows:

$$\begin{aligned} f &\sim \mathcal{GP}\left(m(\boldsymbol{x}),\ k(\boldsymbol{x}, \boldsymbol{x}')\right) \\ m(\boldsymbol{x}) &= \mathbb{E}(f(\boldsymbol{x})) \\ k(\boldsymbol{x}, \boldsymbol{x}') &= \mathbb{E}\left([f(\boldsymbol{x}) - m(\boldsymbol{x})][f(\boldsymbol{x}') - m(\boldsymbol{x}')]\right). \end{aligned}$$

Using the same model as in Section 3.4 with the prior $\boldsymbol{w} \sim \mathcal{N}\left(\mathbf{0}, \alpha^2 I\right)$ we formulate a Gaussian process

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x})$$

which clearly satisfies the requirements to be as such in Definition 3.2. The mean and covariance functions then follow as

$$\begin{aligned} m(\boldsymbol{x}) &= \mathbb{E}(\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x})) = \boldsymbol{\phi}(\boldsymbol{x})^T \mathbb{E}(\boldsymbol{w}) = 0 \\ k(\boldsymbol{x}, \boldsymbol{x}') &= \mathbb{E}\left([\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}) - m(\boldsymbol{x})][\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}') - m(\boldsymbol{x}')]\right) = \boldsymbol{\phi}(\boldsymbol{x})^T \mathbb{E}(\boldsymbol{w}\boldsymbol{w}^T) \boldsymbol{\phi}(\boldsymbol{x}'). \end{aligned}$$

Now we can see that as $\mathbb{E}(\boldsymbol{w}) = \mathbf{0}$,

$$(\mathbb{E}(\boldsymbol{w}\boldsymbol{w}^T))_{ij} = \mathbb{E}(w_i w_j) = \mathbb{E}([w_i - \mathbb{E}(w_i)][w_j - \mathbb{E}(w_j)]) = \mathrm{Cov}(w_i, w_j)$$
$$\implies k(\boldsymbol{x}, \boldsymbol{x}') = \alpha^2 \boldsymbol{\phi}(\boldsymbol{x})^T \boldsymbol{\phi}(\boldsymbol{x}').$$

This would work just fine but does not help in choosing the basis function. For our purpose let's formulate a different kernel called the *squared exponential kernel* which, if we have input points $\boldsymbol{x}^{[1]}, \ldots, \boldsymbol{x}^{[N]}$, is formulated as follows:

$$k(\boldsymbol{x}^{[p]}, \boldsymbol{x}^{[q]}) = \alpha^2 \exp\left(-\frac{\|\boldsymbol{x}^{[p]} - \boldsymbol{x}^{[q]}\|^2}{2\ell^2}\right) \tag{26}$$

for some $\alpha, \ell \in \mathbb{R}$. This choice seems intuitive as it implies that points arbitrarily close together have high covariance which decreases as the points get further away from each other. Although the proof is beyond the scope here, it can be shown that this kernel corresponds to a zero mean Bayesian linear regression model with an infinite number of basis functions [7]. Therefore using this kernel circumvents the choice of $\boldsymbol{\phi}$ completely. In this new kernel, α does not follow from the prior we placed on $\boldsymbol{w}$ previously, but it has the same purpose: to define a scale in which the covariance resides. Also, ℓ intuitively represents a length-scale of the process, so increasing ℓ makes distance between points less influential to their covariance.

Using our previous notation we can again formulate a new covariance matrix K that is essentially a discretisation of the kernel function. For instance with X from (23) we can say that

$$K(X, X)_{ij} = k(\boldsymbol{x}^{[i]}, \boldsymbol{x}^{[j]}).$$

We still possess the same goal here which is to form a conditional distribution for untested points so we need to consider, as in Section 3.3, that an observation y induces some random noise. Indeed

$$y = f(x) + \epsilon$$

where $\epsilon \sim \mathcal{N}\left(0, \sigma^2\right)$. We can then take a finite number of untested and tested points $\boldsymbol{Y_t}$ and $\boldsymbol{Y_u}$ respectively and model the vector

$$\begin{bmatrix} \boldsymbol{Y_t} \\ \boldsymbol{Y_u} \\ \boldsymbol{\epsilon} \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} K(X_t, X_t) & K(X_t, X_u) & 0 \\ K(X_u, X_t) & K(X_u, X_u) & 0 \\ 0 & 0 & \sigma^2 I \end{bmatrix}\right).$$

We can then write

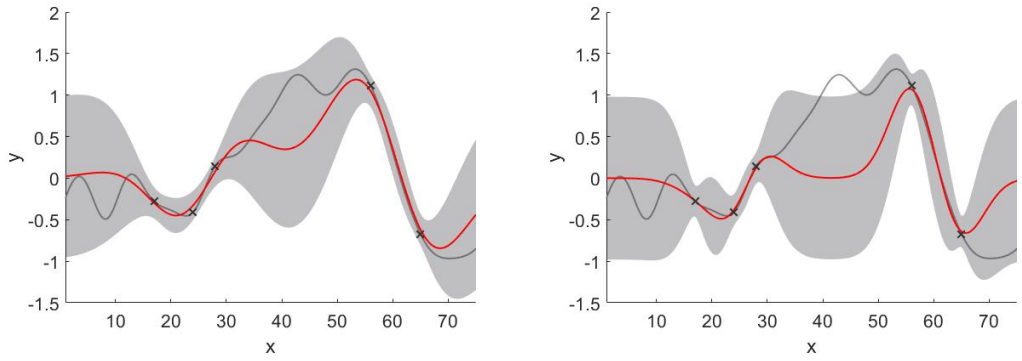
$$\boldsymbol{Y} = \begin{bmatrix} \boldsymbol{Y_t} \\ \boldsymbol{Y_u} \end{bmatrix} = \begin{bmatrix} I & 0 & I \\ 0 & I & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{Y_t} \\ \boldsymbol{Y_u} \\ \boldsymbol{\epsilon} \end{bmatrix}$$

and using Theorem 3.2 we can formulate a distribution of $\boldsymbol{Y}$ as

$$\boldsymbol{Y} \sim \mathcal{N}\left(\mathbf{0}, \Sigma_{\boldsymbol{Y}}\right)$$

where,

$$\Sigma_{\boldsymbol{Y}} = \begin{bmatrix} K(X_t, X_t) + \sigma^2 I & K(X_t, X_u) \\ K(X_u, X_t) & K(X_u, X_u) \end{bmatrix}.$$



(a) $\alpha = 0.5$, $\sigma = 0.1$, $\ell = 7$.

(b) $\alpha = 0.5$, $\sigma = 0.1$, $\ell = 4$.

Figure 6: Posterior drawn from the Gaussian process prior with different values of ℓ. Here the black line shows the objective function; the crosses are sampled points; the red line indicates the posterior mean whilst the shaded region is the mean ± 1.96 times the posterior standard deviation.

Then using Theorem 3.1 we can say that,

$$\begin{aligned} \boldsymbol{Y_u} \mid \boldsymbol{Y_t} = \boldsymbol{y_t} &\sim \mathcal{N}(\boldsymbol{\mu}, \Sigma) \\ \boldsymbol{\mu} &= K(X_u, X_t)^T (K(X_t, X_t) + \sigma^2 I)^{-1} \boldsymbol{y_t} \\ \Sigma &= K(X_u, X_u) - K(X_u, X_t)(K(X_t, X_t) + \sigma^2 I)^{-1} K(X_t, X_u). \end{aligned}$$

So we now have a conditional distribution that we can use for Bayesian optimisation. An example of using a Gaussian process on the same problem as in Figure 5 can be seen in Figure 6. From inspection it is clear how much more flexibility the Gaussian process model gives which will be very important when we eventually apply it to our neural network problem. We can also see the effect that changing ℓ has on the posterior. With the smaller ℓ in Figure 6b the standard deviation increases much faster away from sampled points than in Figure 6a, which is what we would expect from the formulation of the squared exponential kernel (26).

3.6 Acquisition Functions

Now let's turn our attention to choosing a suitable acquisition function as briefly mentioned in Section 3.1. Generally speaking an *acquisition function* is an inexpensive to compute function that evaluates how desirable evaluating f at $\boldsymbol{x}$ would be for our optimisation. From now on let's denote the acquisition function a. To make this possible we can formulate a set of candidate points, say $\mathcal{C}$ to potentially sample and then find the best candidate point

$$\boldsymbol{x}^* = \arg\max_{\boldsymbol{x} \in \mathcal{C}} \left[a(\boldsymbol{x}) \right]$$

based upon the statistical power that the Gaussian process gives us. With all of our work in the previous section this should be entirely feasible but we need a good choice of acquisition function for this to work. The most obvious choice is simply choosing the point with the best mean so,

$$a(\boldsymbol{x}) = \mathbb{E}(f(\boldsymbol{x})).$$

This approach is known as greedy, so choosing the point that is most likely to yield good results. However this turns out to not produce many good results on the global scale and in practice using this function would throttle exploration. Choosing a broadly depends on context, for example if you only had the ability to perform a few iterations you would want to maximise the chance of selecting a good point to evaluate. Contrastingly if you were able to take more time selecting points, a more exploratory approach would be optimal. There are many good choices but the one we are going to consider here is called *expected improvement*. As you would expect, this acquisition function chooses the next point based upon the the expected improvement on the best point previously sampled. The function can be formalised as

$$a(\boldsymbol{x}) = \mathbb{E}(\max(y^* - f(\boldsymbol{x}),\ 0)) \tag{27}$$

where y^* is the maximum sample obtained so far. This expectation also turns out to be feasible to calculate without the need for complex computation and we can formulate an alternative form for expected improvement in Lemma 3.1.

Lemma 3.1. *If we have $f(\boldsymbol{x}) \sim \mathcal{N}\left(\mu,\ \sigma^2\right)$ then,*

$$\mathbb{E}(\max(y^* - f(\boldsymbol{x}),\ 0)) = (y^* - \mu)\Phi\left(\frac{y^* - \mu}{\sigma}\right) + \sigma\phi\left(\frac{y^* - \mu}{\sigma}\right)$$

where Φ and ϕ are the standard cumulative and probability density functions respectively.

Proof. First notice that $f(\boldsymbol{x}) = \mu + \sigma Z$ where $Z \sim \mathcal{N}(0,\ 1)$. We can then use the Law of the Unconscious Statistician [1] to say that,

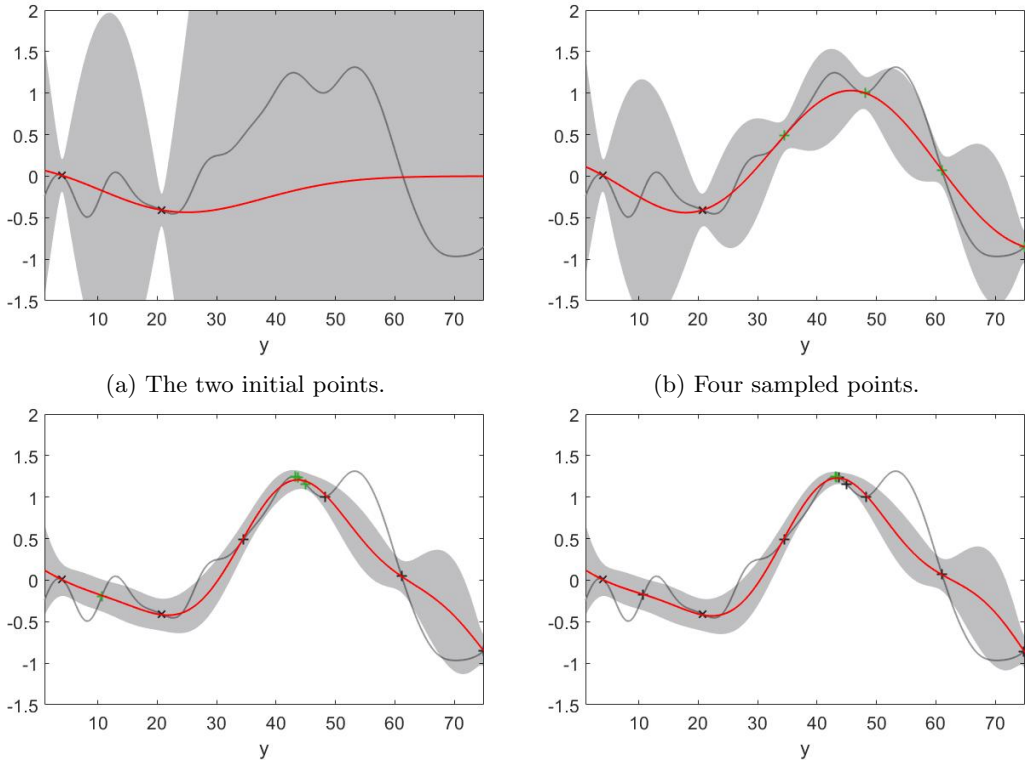
$$\mathbb{E}(\max(y^* - f(\boldsymbol{x}),\ 0)) = \mathbb{E}(\max(y^* - \mu - \sigma Z,\ 0)) = \int_{-\infty}^{\infty} \max(y^* - \mu - \sigma z,\ 0)\phi(z)\ dz$$

Now if $y^* - \mu - \sigma z < 0$ we have that $\max(y^* - \mu - \sigma z,\ 0) = 0$ so if we define $L := \frac{y^* - \mu}{\sigma}$ we can restrict the integral,

$$\begin{aligned}
\mathbb{E}(\max(y^* - f(\boldsymbol{x}),\ 0)) &= \int_{-\infty}^{L} \max(y^* - \mu - \sigma z,\ 0)\phi(z)\ dz \\
&= \int_{-\infty}^{L} (y^* - \mu - \sigma z)\phi(z)\ dz \\
&= (y^* - \mu)\int_{-\infty}^{L} \phi(z)\ dz - \sigma\int_{-\infty}^{L} z\phi(z)\ dz \\
&= (y^* - \mu)\Phi(L) - \sigma\int_{-\infty}^{L} z\frac{1}{\sqrt{2\pi}}\exp\left(-\frac{z^2}{2}\right)\ dz
\end{aligned} \tag{28}$$

by the definition of Φ and ϕ. Using the substitution $u = -\frac{z^2}{2}$ we can reformulate the remaining integral,

$$\begin{aligned}
\sigma\int_{-\infty}^{L} z\frac{1}{\sqrt{2\pi}}\exp\left(-\frac{z^2}{2}\right)\ dz &= -\frac{\sigma}{\sqrt{2\pi}}\int_{-\infty}^{-\frac{L^2}{2}} \exp(u)\ du \\
&= -\frac{\sigma}{\sqrt{2\pi}}\exp\left(-\frac{L^2}{2}\right) \\
&= -\sigma\phi(L)
\end{aligned}$$



(a) The two initial points.

(b) Four sampled points.

(c) Eight sampled points.

(d) Twelve sampled points.

Figure 7: Visualisation of the posterior distribution at different stages of the Bayesian optimisation algorithm. The black line is the objective function; the red line indicates the posterior mean whilst the shaded areas represent the mean ± 1.96 times the posterior standard deviation. The crosses are the initially sampled points (pre-optimisation) and the pluses subsequently sampled points (green pluses being newly sampled).

Substituting this in to (28) yields,

$$\mathbb{E}(\max(y^* - f(\boldsymbol{x}),\ 0)) = (y^* - \mu)\Phi(L) - \sigma\phi(L) = (y^* - \mu)\Phi\left(\frac{y^* - \mu}{\sigma}\right) - \sigma\phi\left(\frac{y^* - \mu}{\sigma}\right)$$

as required. □

We now have a suitable acquisition function which we can use to complete our Bayesian optimisation algorithm in Section 3.1. Using this and all the work in the previous sections we can visualise the algorithm in Figure 7. As you can see the algorithm does not take long to capture the behaviour of the objective function and although the interval in which the maximum resides was initially unexplored, most of the newly sampled points occurred there. We can also see that the standard deviation decreases very quickly and clearly is smaller close to the sampled points, which is consistent with what we would expect from (26). We can now apply this to Problem 2 and hopefully it can find us a better optimum solution than could have been obtained through experimentation.

It is worth noting that although expected improvement is widely used in practice, it is often too greedy to be optimal. Variations to this acquisition function have been formulated and shown to have a dramatic improvement on results obtained with the original [6]. However, this is beyond the scope here and for simplicity we will use standard expected improvement acquisition.

4 Optimising a Neural Network

Up until now all of our work towards implementing Bayesian optimisation has been non-concrete, but now we can aim to use it to finally solve Problem 2. First recall that we are using the validation accuracy as a measure of our network's performance, so a function f which computes validation accuracy is our objective function. This is an intuitive and easy to work with choice, but it might be worth considering others. A well formulated metric is the OTMANN distance [4] which takes into account how layers are connected, along with the computation time at each layer which we are not considering here. We can consider vectors of hyperparameters as our input points and to aid understanding we will establish an explicit selection of parameters, though adding more would be trivial. So let's fix a maximum number of layers L that the network could have and look at a blueprint for an input vector:

$$\boldsymbol{x} = [n_1, \dots, n_L, E, \eta]^T. \tag{29}$$

Here we are using the notation from Section 2.1 so n_l denotes the number of neurons in layer l, E the number of training epochs and η the learning rate. Immediately there are a couple of issues to address. The first is how to optimise the number of layers, however if we pose that $n_l = 0$ means there is no layer at layer l (as strange as that may sound), this problem fixes itself. Another complication comes when we start to use the Gaussian process formulated in Section 3.5 where we have various prior parameters α, σ, ℓ. In our input vector $\boldsymbol{x}$ each hyperparameter has a different scale so these prior parameters would have to be scaled accordingly. To bypass implementing this we can simply scale $\boldsymbol{x}$ to a unit scale. So if we impose intervals for each hyperparameter (which would need to be chosen beforehand),

$$0 \leq n_l \leq n_{\max}; \qquad E_{\min} \leq E \leq E_{\max}; \qquad \eta_{\min} \leq \eta \leq \eta_{\max},$$

then we can use an alternative input vector:

$$\boldsymbol{x} = \left[\frac{n_1}{n_{\max}}, \dots, \frac{n_L}{n_{\max}}, \frac{E - E_{\min}}{E_{\max} - E_{\min}}, \frac{\eta - \eta_{\min}}{\eta_{\max} - \eta_{\min}}\right]^T. \tag{30}$$

We just need to ensure that we use (30) in our Bayesian optimisation but pass (29) to our neural network. A final issue is that all hyperparameters here, save the learning rate, are integers though some careful rounding whilst training the network will work just fine.

Now let's define the set of all combinations of (unscaled) hyperparameters

$$\mathcal{X} = \left\{[n_1, \dots, n_L, E, \eta]^T \in \mathbb{R}^{L+2} : 0 \leq n_l \leq n_{\max},\, E_{\min} \leq E \leq E_{\max},\, \eta_{\min} \leq \eta \leq \eta_{\max}\right\}$$

and we can choose a set of candidate points $\mathcal{C} \subset \mathcal{X}$ to model at each iteration of our optimisation, with $n_c := |\mathcal{C}|$. Also recall our input matrix X (23) which we can reformulate slightly to refer to candidate points as

$$X = \begin{bmatrix} X_t \\ X_c \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}_{\boldsymbol{t}}^{[1]} & \cdots & \boldsymbol{x}_{\boldsymbol{t}}^{[n]} & \boldsymbol{x}_{\boldsymbol{c}}^{[1]} & \cdots & \boldsymbol{x}_{\boldsymbol{c}}^{[n_c]} \end{bmatrix}^T$$

where $\boldsymbol{x}_{\boldsymbol{t}}^{[i]} \in \mathcal{X}$, $\boldsymbol{x}_{\boldsymbol{c}}^{[j]} \in \mathcal{C}\ \forall\ i \in \{1, \dots, n\}$, $j \in \{1, \dots, n_c\}$ can denote sampled points and candidate points respectively with n sampled points. We will make sure that before we try

to choose candidate points we already have some observed data which can just be randomly sampled. Then using our Gaussian process to model f as in Section 3.5 we can say that,

$$\begin{aligned} \boldsymbol{Y_c} \mid \boldsymbol{Y_t} = \boldsymbol{y_t} &\sim \mathcal{N}(\boldsymbol{\mu}, \Sigma) \\ \boldsymbol{\mu} &= K(X_c, X_t)^T (K(X_t, X_t) + \sigma^2 I)^{-1} \boldsymbol{y_t} \\ \Sigma &= K(X_c, X_c) - K(X_c, X_t)(K(X_t, X_t) + \sigma^2 I)^{-1} K(X_t, X_c) \end{aligned}$$

where $\boldsymbol{Y_c}$ can denote the value of f at the candidate points. We can also model the output at each candidate point $(\boldsymbol{Y_c})_i$ as univariate normal

$$(\boldsymbol{Y_c})_i \sim \mathcal{N}(\mu_i, \Sigma_{ii})$$

and we can find the maximum sample from our observed data

$$y^* = \max_{1 \leq i \leq n} (\boldsymbol{y_t})_i$$

at each iteration. This then enables us to use our acquisition function (27) as follows:

$$a((\boldsymbol{x_c})_i) = (y^* - \mu_i)\Phi\left(\frac{y^* - \mu_i}{\Sigma_{ii}}\right) - \Sigma_{ii}\phi\left(\frac{y^* - \mu_i}{\Sigma_{ii}}\right).$$

Note that the ϕ here is the pdf of a normal distribution, not a basis function. This gives us everything we need to start optimising our neural network so let's formulate a full algorithm to do so (Algorithm 4). We can now consider an example problem similar to that seen in

Algorithm 4 Optimising Neural Network Hyperparameters

Input: A function $f : \mathcal{X} \to [0, 1]$ that outputs validation accuracy of a neural network; intervals for each hyperparameter; positive prior parameters α, σ, $\ell \in \mathbb{R}$; the number of points to initially sample N_{initial}; the number of candidate points to consider at each iteration n_c and the number of iterations $N_{\text{iterations}}$

Randomly select N_{initial} input points in the interval $[0, 1]$
Sample f at each of these points post-scaling

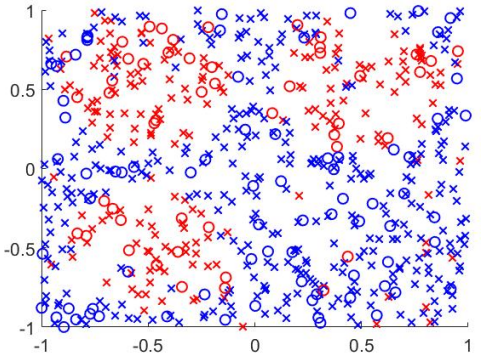
for $i = 1, \ldots, N_{\text{iterations}}$ **do**

    Randomly choose n_c candidate points in the interval $[0, 1]$ for each hyperparameter
    Assemble the matrix X with all sampled and candidate points
    Compute the mean $\boldsymbol{\mu} = K(X_c, X_t)^T (K(X_t, X_t) + \sigma^2 I)^{-1} \boldsymbol{y_t}$ and covariance matrix
        $\Sigma = K(X_c, X_c) - K(X_c, X_t)(K(X_t, X_t) + \sigma^2 I)^{-1} K(X_t, X_c)$
    $y^* = \max_{1 \leq j \leq i} [(\boldsymbol{y_t})_j]$
    $\boldsymbol{x}^* = \arg\max_{\boldsymbol{x} \in \mathcal{C}} [a(\boldsymbol{x})]$
    Sample f at $\boldsymbol{x}^*$ post-scaling

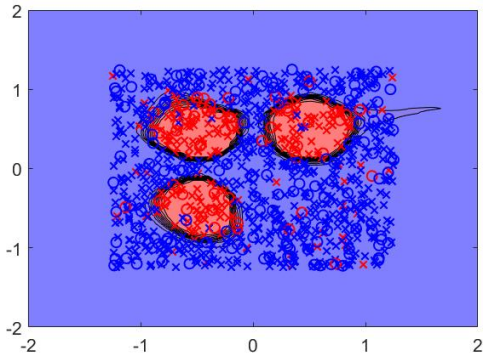
end for

return the hyperparameters corresponding to the best validation accuracy

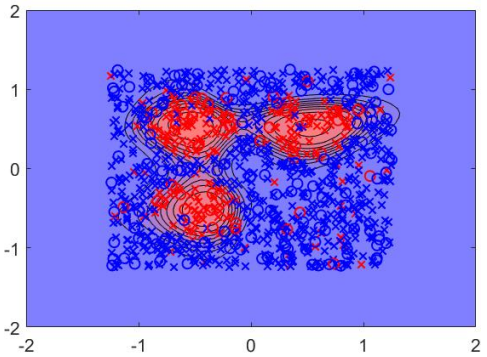
Section 2 (Figures 3 and 4) and try to optimise a neural network to solve it, shown in



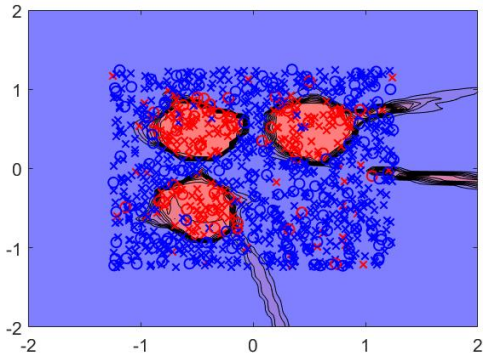
(a) The training data, represented by crosses; and the test data, represented by circles.



(b) $\eta = 0.0468$, the optimum value. Corresponds to a validation accuracy of 0.895.



(c) $\eta = 0.005$. Corresponds to a validation accuracy of 0.865.



(d) $\eta = 0.5$. Corresponds to a validation accuracy of 0.87.

Figure 8: Neural network classification based on an optimal architecture and number of training epochs.

Figure 8. Unfortunately the optimisation seems to always favour networks with a larger architecture. This could be due to an ideal network being much larger than could be tested here, due to computational limitations. Adding some sort of computational cost to our acquisition function could help to optimise these smaller neural networks as briefly mentioned at the end of Section 3.6, but we will not explore this here. The results do seem to show an interesting optimisation of the learning rate however, and there is a noticeable drop in accuracy below and above the optimised parameter value which seems consistent when repeating the tests and can be seen in Figures 8c and 8d.

4.1 Concluding Remarks

In this paper we have thoroughly introduced the theory behind neural networks and then started to approach our main objective which was solving Problem 2. Using Bayesian optimisation with Gaussian process priors we were able to formulate a full algorithm to optimise the architecture of a neural network, along with some other important hyperparameters. The final results seemed to favour larger architectures due to computational limitations but

worked well on small parameters. Something beyond the scope here was optimising our prior parameters α, σ, ℓ which would make the neural network optimisation completely autonomous [7]. Regardless the Bayesian optimisation approach provides a computationally efficient way to choose neural network hyperparameters which removes the need for experimentation or time consuming parameter searches. Actual implementations of the main algorithms, Algorithm 3 and 4 can be seen in Appendix A.

References

[1] Joseph Blitzstein. *Introduction to Probability*. CRC Press, 2015.

[2] Catherine F. Higham and Desmond J. Higham. "Deep Learning: An Introduction for Applied Mathemticians". In: (Jan. 19, 2018).

[3] J.-S. Roger Jang. *Matrix Inverse in Block Form*. Mar. 21, 2001. URL: `http://www.cs.nthu.edu.tw/~jang/book/addenda/matinv/matinv/` (visited on 04/04/2019).

[4] Kirthevasan Kandasamy et al. "Neural Architecture Search with Bayesian Optimisation and Optimal Transport". In: (Feb. 11, 2018).

[5] Siyuan Ma, Raef Bassily, and Mikhail Belkin. "The Power of Interpolation: Understanding the Effectiveness of SGD in Modern Over-parametrized Learning". In: (June 14, 2018).

[6] Chao Qin, Diego Klabjan, and Daniel Russo. "Improving the Expected Improvement Algorithm". In: (2017).

[7] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. The MIT Press, Nov. 23, 2005.

[8] Bobak Shahriari et al. "Taking the Human Out of the Loop: A Review of Bayesian Optimization". In: (Dec. 10, 2015).

[9] Jasper Snoek and Hugo Larochelle. "Practical Bayesian Optimization of Machine Learning Algorithms". In: (2012).

[10] Ruye Wang. *Inverse and determinant of partitioned symmetric matrix*. Nov. 14, 2006. URL: `http://fourier.eng.hmc.edu/e161/lectures/gaussianprocess/node6.html` (visited on 04/04/2019).

[11] Robert L. Wolpert. *STA 114: Statistics*. Feb. 17, 2011. URL: `https://www2.stat.duke.edu/courses/Spring11/sta114/lec/114mvnorm.pdf` (visited on 04/04/2019).

Appendices

‘

A Code

This is a working MATLAB implementation of Algorithm 3.

```
function [W, b] = trainneuralnetwork(X, y, hiddenNeuronPattern,
    nEpochs, learningRate)
% Uses back propogation to train a neural network
% Inputs:   X - a d by n matrix of training data
%           y - an c by n matrix of corresponding classifications
%           hiddenNeuronPattern - an (l - 2) length vector
    specifiying how many neurons
%               should exist at each layer from layer 2 to l - 1
%           nEpochs - the number of epochs to train with
%           learningRate - the learning rate
% Outputs:  W - a cell array of weights matrices for each layer
%           b - a cell array of biases for each layer
% Note that the output cell arrays contain an empty first element
    for the
% input layer

dIn = size(X, 1);
dOut = size(y, 1);
nPoints = size(X, 2);

neurons = [dIn, hiddenNeuronPattern(hiddenNeuronPattern ~= 0),
    dOut];
nLayers = length(neurons);

W = cell(1, nLayers);
b = cell(1, nLayers);

% Randomly initialise the weights and biases
for i = 2:nLayers
    W{i} = randn(neurons(i), neurons(i - 1));
    b{i} = randn(neurons(i), 1);
end

a = cell(1, nLayers);
delta = cell(1, nLayers);

for epoch = 1:nEpochs

    % Shuffle the training data
    order = randperm(nPoints);
```

```
    % Use back propogation to update the weights and biases
    for i = 1:nPoints
        a{1} = X(:, order(i));

        for j = 2:nLayers
            a{j} = activateneuron(W{j}, a{j-1}, b{j});
        end

        aTemp = a{nLayers};
        delta{nLayers} = aTemp.*(1 - aTemp).*(aTemp - y(:, order(i
            )));

        for j = nLayers-1:-1:2
            delta{j} = a{j}.*(1 - a{j}).*(W{j + 1}.' * delta{j +
                1});
        end

        for j = 2:nLayers
            W{j} = W{j} - learningRate*delta{j}*a{j - 1}.';
        end

        for j = 2:nLayers
            b{j} = b{j} - learningRate*delta{j};
        end
    end
end

end

function a = activateneuron(W, x, b)
% Activates a neuron by computing sigm(W x + b)
% Inputs:   W - a matrix of weights for a particular neuron
%           x - the input vector
%           b - a bias vector for the neuron
% Ouputs:   a - sigm(W*x + b)

a = sigm(W*x + b);

end

function y = sigm(x)
% A special case of the logistic function
% Output:   y = 1/(1+e^(-x))

    y = 1./(1+exp(-x));

end
```

This is a MATLAB function that implements Theorem 3.1 to compute a posterior mean and covariance matrix based upon a prior mean and covariance.

```
function [posteriorMean, posteriorCov] = conditionalmvn(priorMean,
    priorCov, x_1)
% Computes the posterior mean and covariance matrix for a
   conditional
% multivariate normal
% Inputs:   priorMean - the prior mean
%           priorCov - the prior covariance matrix
%           x_1 - the given points
% Outputs:  posteriorMean - the conditional mean
%           posteriorCov - the conditional covariance matrix

% Split the inputs into blocks
n = length(x_1);
I = [n, length(priorMean) - n];
m = mat2cell(priorMean, I);
c = mat2cell(priorCov, I, I);

% Calculate the posterior
posteriorMean = m{2} + c{2,1}*(c{1,1}\(x_1 - m{1}));
posteriorCov = c{2,2} - c{2,1}*(c{1,1}\c{1,2});

end
```

This is a MATLAB function that uses `conditionalmvn.m` to compute the posterior from a zero mean Gaussian process using the squared exponential function (26).

```
function [posteriorMean, posteriorCov] = gaussianprocesscondmvn(
    X_t, X_u, y_t, alpha, sigma, ell)
% Calculate the conditional posterior mean and covariance matrix
    using the
% squared exponential kernel
% Inputs:    X_t - a matrix of tested points, stored in columns
%            X_u - a matrix of untested points, stored in columns
%            y_t - the output corresponding to the tested input
    points
%            alpha - scale of the covariance
%            sigma - the standard deviation of the error
%            ell - the effect that distance has on covariance
% Outputs:   posteriorMean - the conditional posterior mean
%            posteriorCov - the conditional posterior covariance
    matrix

n = size(X_t, 2);
N = n + size(X_u, 2);

X = [X_t, X_u]';

% Calculate the prior mean and covariance
priorMean = zeros(N, 1);
priorCov = zeros(N);
for i = 1:N
    for j = 1:N
        priorCov(i,j) = alpha^2*exp(-(norm(X(i,:) - X(j,:))^2)/(2*
            ell^2));
    end
end
for i = 1:n
    priorCov(i,i) = priorCov(i,i) + sigma^2;
end

% Compute the posteriors
[posteriorMean, posteriorCov] = conditionalmvn(priorMean, priorCov
    , y_t);

end
```

This is a working MATLAB implementation of general Bayesian optimisation that can be applied to optimising neural network hyperparameters with f as a network's validation accuracy as in Algorithm 4.

```
function [x, y] = bayesianopt(f, range, alph, sigma, ell, nReveal,
    nCandidates, nIterations)
% Attempts to optimse a real gaussian process
% Inputs:   f - the function to optimise
%               (should take input of an 'nParams' by 1 matrix and
    output
%               a real number in [0,1])
%           range - an 'nParams' by 2 matrix defining the range to
    search
%               for each parameter
%           alpha - the standard deviation of the weights
%           sigma - the standard deviation of the residuals
%           ell - measure of effect that close points have on
   eachother
%           nReveal - the number of parameter sets to initially
   randomly
%               reveal
%           nCandidates - the number of parameter sets to consider
    each
%               iteration
%           nIterations - the number of iterations to perform
% Ouputs:   x - the best hyperparameters
%           y - the best validation accuracy

nParams = size(range, 1);

x_t = zeros(nParams, nIterations + nReveal);
y_t = zeros(nIterations + nReveal, 1);

% Sample at the initial points
for i = 1:nReveal
    x_t(:,i) = rand(nParams, 1);
    y_t(i) = f(scaleinput(x_t(:,i), range));
end

for i = nReveal:nReveal + nIterations - 1

    % Choose candidate points
    candidatePoints = rand(nParams, nCandidates);
    [posteriorMean, posteriorCov] = gaussianprocesscondmvn(x_t
        (:,1:i), candidatePoints, y_t(1:i), alph, sigma, ell);

    % Expected improvement
    yMax = max(y_t(1:i));
    meanDist = posteriorMean - yMax;
```

```
    sd = sqrt(diag(posteriorCov));
    expectedImprovement = meanDist.*normcdf(meanDist./sd) + sd.*
        normpdf(meanDist./sd);
    [~, reveal] = max(expectedImprovement);

    % Sample at the chosen point
    x_t(:,i+1) = candidatePoints(:, reveal);
    y_t(i+1) = f(scaleinput(x_t(:,i+1), range));
end

% Choose the best point sampled
[y, iBest] = max(y_t);
x = x_t(:,iBest);

end

function scaledInput = scaleinput(x, range)
% Scales uniform input
% Inputs:    x - an N by 1 input vector where each element is in
    [0, 1]
%           range - an N by 2 vector of ranges
% Outputs:  scaledInput - an N by 1 vector where the ith element
    is the
%               ith element of 'x' scaled between the first and
    second
%               elements of the ith row of 'range'

scaledInput = (range(:,2) - range(:,1)).*x + range(:,1);

end
```